2 Enkla datatyper

Att skriva ett program innebär att man ska tillverka en plan för att bearbeta data eller information på något sätt. Programmering handlar om *data och bearbetning av data.* Data kan vara av enkel typ som tal och tecken eller av mer sammansatt typ som text, tabeller, ljud, bilder etc. När det gäller att tillfälligt spara undan data under bearbetningen använder man sig av *variabler*.

2.1 Variabler

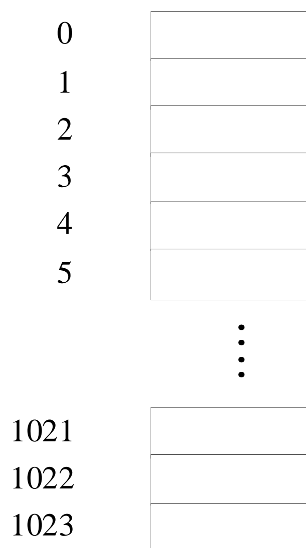
Variabler använder man sig av för att under programkörningen spara data eller information i primärminnet. En variabel har *typ, namn och värde*.

Ex: En variabel av typen int med namnet tal och initialvärdet 2314 skapas enligt:

```
int tal = 2314;
```

Med namnet på variabeln hänvisar man till en speciell plats i primärminnet. Primärminnet kan liknas vid ett stort antal postfack. Storleken på varje fack är 1 byte, vilket (nästan alltid) motsvarar 8 bitar. För att kunna stoppa in data i eller läsa data från rätt fack har varje fack en unik adress.

Ex: Ett minne på 1 KB = 1024 B kan ges adresser enligt:

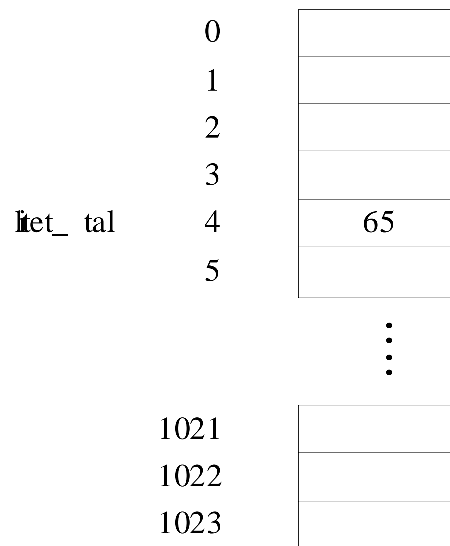


Det är inte lätt att hålla reda på alla fack och deras adresser. Därför har man i högnivåspråken infört variabelbegreppet. Man definierar en variabel med ett unikt namn som kompilatorn automatiskt kopplar ihop med av typen angivet utrymme i minnet. På detta sätt slipper man att hålla reda på exakt var i minnet data finns. Man använder istället variabelnamnet då man vill komma åt data.

Ex: Har man i sitt program definierat variabeln litet_tal av typen char och tilldelat den värdet 65 enligt:

```
char litet_tal;
litet_tal = 65;
```

kan minnesbilden se ut som:



Istället för att vara tvungen att komma ihåg att litet_tal finns på adressen 4 använder man variabelnamnet litet_tal för att komma åt data enligt exempelvis:

```
litet_tal = 34;                /* nytt värde 34 */
litet_tal = litet_tal + 12;    /* nytt värde blir 46 */
```

Variabelnamnet får innehålla bokstäver, siffror och understrykningstecken. Namnet måste dock börja med en bokstav eller ett understrykningstecken. Hur många tecken man kan ha i variabelnamnen beror på vilken kompilator man använder. Oftast får man dock använda åtminstone 31 tecken.

Ex: x_koordinat
minsta_tal
max_tal_3

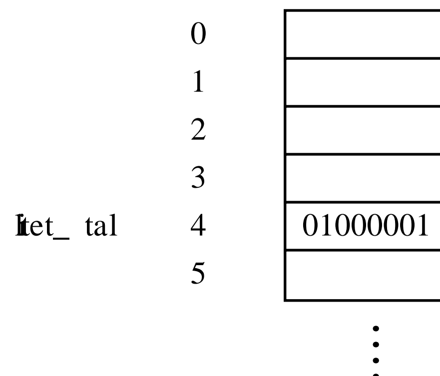
OBS! Man får ej ha de svenska bokstäverna Å, å, Ä, ä, Ö, ö i variabelnamnen.

OBS! Använd variabelnamn som utsäger något. Var inte rädd för att använda långa variabelnamn. En programtext skrivs en gång men läses betydligt fler gånger. Använd understrykningstecknet för att göra långa namn tydligare.

En variabels värde skriver vi i vårt högnivåspråk på det sätt som är brukligt när man skriver tal, tecken, text etc. Kompilatorn måste dock koda vårt data på något sätt till ettor och nollor. Tal kodas i vanlig binärkod där 2 används som bas istället för 10.

Ex: Ange hur de enskilda bitarna ser ut i minnes-facket litet_tal ovan

$$65 = 0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 01000001$$



Vad kan man då spara för variabelvärden i ett minnesfack bestående av 8 bitar? Detta beror naturligtvis på vilken form av kod som används. Allmänt kan man spara 2*2*2*2*2*2*2*2 = 256 olika kodkombinationer i 8 bitar eftersom varje bit kan anta två värden. Detta innebär att man kan spara 256 olika heltal.

2.1.1 Heltal

Ex: Hur stora positiva heltal kan man spara i 1 byte om vanlig binärkod används?

0	00000000
1	00000001
2	00000010
3	00000011
.	
.	
255	11111111

Alltså kan man spara tal från 0 till 255.

Vill man även använda negativa tal måste man på något sätt i koden markera att talet är negativt. Man kan exempelvis låta den första biten markera tecknet (plus eller minus), och då finns det bara plats för 128 olika tal i de resterande 7 bitarna. Den vanligaste koden för negativa tal är dock tvåkomplementmetoden.

Ex: Vilka heltal kan man spara i 1 byte om kodformen tvåkomplementmetoden används för negativa tal?

0	00000000
1	00000001
.	
.	
127	01111111
-128	10000000
-127	10000001
.	
.	
-1	11111111

Alltså kan man spara tal mellan -128 och 127. Tvåkomplementmetoden utgår från binärkoden för det positiva talet och byter alla ettor mot nollor och nollor mot ettor. Efter addition med 1 har man fått den nya koden.

Vad gör man om man vill använda större tal än vad som ryms i 8 bitar? Man utnyttjar naturligtvis flera minnesfack och får då tillgång till flera bitar och ett större antal kombinationer.

Ex: Hur stora positiva heltal kan man spara i 16 bitar?

8 bitar rymmer talen 0 till $2^8 - 1 = 255$

16 bitar rymmer talen 0 till $2^{16} - 1 = 65535$

Ex: Hur stora heltal kan man spara med tvåkomplementmetoden i 16 bitar?

8 bitar rymmer talen $-2^7 = -128$ till $2^7-1 = 127$

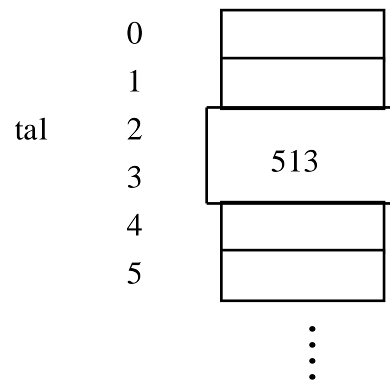
16 bitar rymmer talen $-2^{15} = -32768$ till $2^{15}-1 = 32767$

Det är den angivna datatypen som bestämmer hur många minnesfack och vilken kodning som ska användas för den definierade variabeln. Med typen short anger man exempelvis att man ska använda binärkodade tal i 16 bitar. (Antalet bitar för de olika datatyperna kan dock variera något mellan olika kompilatorer.)

Ex: Variabeldefinitionen

```
short tal = 513;
```

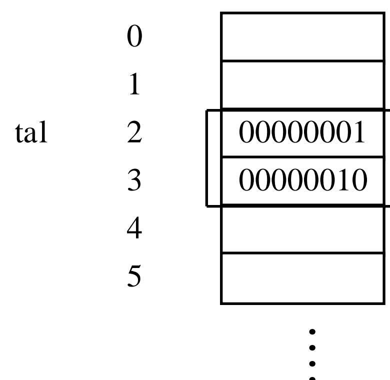
ger minnesbilden:



och eftersom

513 = 0000 0010 0000 0001 = 0 2 0 1 (hexadecimalt)

blir bitmönstret i minnet:



Vilka typer av variabler som finns beror oftast på vilken dialekt av C man har. Exempelvis kan det finnas följande datatyper att tillgå för hela tal:

unsigned char	8 bitar	0 .. 255
char	8	-128 .. 127
unsigned short	16	0 .. 65 535
short	16	-32 768 .. 32 767
unsigned int	32	0 .. 4 294 967 295
int	32	-2 147 483 648 .. 2 147 483 647

(Dessa bitantal gäller i de flesta vanliga system. C-standarden tillåter dock att kompilatorer har andra storlekar på datatyperna, inom vissa gränser.)

2.1.2 Tecken

När det gäller tecken använder man i högnivåspråket de vanliga beteckningarna för bokstäver och tecken omgivna av enkla apostrofer som i 'A', '+', '7'. Tecknen lagras i form av teckenkoder, som är små heltal. Den mest kända teckenkodningen kallas ASCII. ASCII är en sjubitarskod, och kan alltså lagra 128 olika tecken. Svenska tecken som Å, Ä och Ö finns inte med. Moderna system kräver fler tecken, och de flesta kompilatorer använder därför 8-bitarstecken, och då kan man hantera 256 olika tecken.

Ex: Har man i sitt program definierat variabeln tecken av typen char kan man tilldela den bokstaven A enligt:

```
char tecken;
tecken = 'A';
```

Kompilatorn kodar teckenvärdet till teckenkoden, som för tecknet 'A' är 65. Variabeln tecken är av heltalstypen char och kan ses som:

tecken	'A'

eller

tecken	65

Ex: Skriv ett program som läser in ett tecken och skriver ut tecknet och dess ASCII-kod.

```
#include <stdio.h>

int main()
{
   char tecken;

   /* läs tecken */
   printf("Ge ett tecken : ");
   scanf("%c", &tecken);

   /* skriv tecken och dess teckenkod */
   printf("Tecknet %c har teckenkoden %d\n", tecken, tecken);

   return 0;
}
```

Matar man in bokstaven A blir utskriften:

Tecknet A har teckenkoden 65

OBS! Med omvandlingspecifikationen %c i utskriftssatsens formatsträng anges att minnesutrymmets ettor och nollor ska tolkas som koden för ett tecken och skivas ut som ett tecken . Omvandlingsspecifikationen %d anger att samma minnesutrymme ska tolkas som koden för ett tal och skivas som decimalt tal.

2.1.3 Reella tal

Alla tal är inte hela tal. När man ska hantera data i form av reella tal i datorn används oftast koder där man skriver om talet som en potens av 2 och sedan i minnet sparar mantissan (talet framför potensen) och exponenten i binärkod.

Ex: Talet 5.5 kan exempelvis skrivas om som:

$$5.5 = 2^2 + 2^0 + 2^{-1} = (1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4}) \cdot 2^3$$

Här sparar man mantissans koefficienter 1011 och exponenten 3 = 11 i binär form. Man ser att ju fler decimaler eller ju större tal desto fler bitar behövs. Även tecknet på talen kräver en bit.

I de flesta miljöer finns följande typer för reella tal med angivna antalet bitar och storleksordningar:

float	32	$\pm 1.2 \cdot 10^{-38}$	..	$\pm 3.4 \cdot 10^{38}$	(7 värdesiff)
double	64	$\pm 2.2 \cdot 10^{-308}$	..	$\pm 1.8 \cdot 10^{308}$	(15 värdesiff)

Det är `double` som är den ”vanliga” flyttalstypen i C. Typen `float` brukar man bara använda om man av någon anledning måste spara plats.

Ex: Skriv ett program som frågar efter en kropps massa och hastighet och beräknar och skriver ut dess kinetiska energi som är lika med massa*hastighet*hastighet/2.

```
#include <stdio.h>

int main()
{
   double massa, hastighet, kinetisk_energi;

   /* läs massa */
   printf("Massa = ");
   scanf("%lf", &massa);

   /* läs hastighet */
   printf("Hastighet = ");
   scanf("%lf", &hastighet);

   /* beräkna och skriv ut kinetisk energi */
   kinetisk_energi = massa*hastighet*hastighet/2;
   printf("Kinetisk energi: %f\n", kinetisk_energi);

   return 0;
}
```

`"%lf"` i `scanf`-anropen kan läsas som ”long float”, dvs flyttal med dubbel precision. I `printf` ska man dock skriva `"%f"`. (Fråga läraren om varför!)

2.1.4 Egenuppräknade

Man kan även definiera egna variabeltyper där man ger namn åt varje enskilt värde.

Ex: Definiera en egenuppräknad variabel som ska kunna anta tillståndsvärdena upp, ner och stilla för en hiss.

```
/* definiera en egen datatyp enum hiss */
enum hiss { upp, ner, stilla };

/* definiera en variabel med namnet a_hiss */
enum hiss a_hiss;

/* tilldela a_hiss värdet stilla */
a_hiss = stilla;
```

För egendefinierade typer används vanlig binärkod där det först uppräknade värdet får koden 0 om inget annat anges.

Ex: Definiera en egen variabeltyp som ska kunna ha värden i form av månadsnamn och där månadsnamnen ska motsvaras av motsvarande månadsnummer. Månaden januari ska alltså ha värdet 1 och december värdet 12.

```
/* definiera en egen datatyp enum manad */
enum manad { januari = 1, februari, mars, april, maj, juni,
             juli, augusti, september, oktober, november,
             december };

/* variabel mm av typen enum manad och några andra */
enum manad mm;
int yy, dd, datum;

/* tilldela värden */
yy = 2001;
mm = september;
dd = 14;

/* skriv ut datum på formen 010914 */
datum =(yy - 2000)*10000 + mm*100 + dd;
printf("%06d", datum); /* utfyllnad med 0 till 6 positioner */
```

Egenuppräknade värden används för att göra koden lättare att läsa. För den som ska tyda koden då den ska underhållas eller ändras är det lättare att förstå sammanhanget om det står september istället för bara talet 9.

2.2 Konstanter

Variabler ska kunna ändra sina värden under programkörningen genom att man utför vissa operationer på dessa. Man kan exempelvis läsa in ett nytt värde till en variabel eller man kan multiplicera det med 2 etc.

Ibland vill man se till att ett minnesutrymmes värde ej får ändras under programkörning. Man kan då med *const* framför typnamnet kvalificera eller märka detta minnesfack. Det värde som minnesfacket ska ha måste man ge direkt vid definitionen genom initiering.

Ex: Man ska ha ett program med priser och moms.

```
const double moms = 25.0;
const int pris_per_st = 50;
int st;
double pris, pris_med_moms;

printf("Ge antal : ");
scanf("%d", &st);

pris = pris_per_st*st;
pris_med_moms = pris + pris*moms/100;
```

Varför ska man använda ordet moms eftersom man lika gärna kan skriva 25.0 istället. Fördelen med konstanter är att man har dessa samlade i början av programmet. Ändras momsen räcker det med att ändra på ett ställe istället för att leta upp alla ställen där moms-värdet finns i programmet.

Ett annat sätt att ge namn åt konstanter som ofta används speciellt i äldre C-program är att definiera ett makro. Ett makro är ett direktiv till kompilatorn att vid förkompileringen byta ut alla makro-namn mot motsvarande värde.

Ex: Skriv ett program som läser in en radie och beräknar volymen av ett klot med denna radie. Använd ett makro för PI.

```
#include <stdio.h>
#define PI 3.14159

int main()
{
   double radie;

   printf("Ge radie: ");
   scanf("%lf", &radie);

   printf("Klotvolym = %f\n", 4*PI*radie*radie*radie/3);

   return 0;
}
```

PI byts ut mot 3.14159 vid förkompileringen.

I vissa sammanhang i vissa C-kompilatorer fungerar det inte med `const`-deklarerade konstanter, utan där måste man använda makron.

2.3 In- och utmatning

Ett progam som ska bearbeta data eller information måste på något sätt få tillgång till denna data via någon kanal som exempelvis tangentbord, sekundärminne, port, ljudkort, video-kamera etc. Processen att hämta in data till ett program kallas för *inmatning*.

Efter bearbetningen måste man också kunna presentera resultatet på någon enhet som exempelvis bildskärm, sekundärminne, printer, port, ljudkort etc. Processen att skicka ut data eller presentera data kallas för *utmatning*.

Vanliga enkla in- och utmatningar till variabler av enkla typer gör man via tangentbordet och skärmen. I C finns i *stdio.h* funktionerna *scanf* och *printf* som sköter in- resp utmatning av enkla variabler. Dessutom finns funktionerna *getchar* och *putchar* för in- resp utmatning av tecken.

2.3.1 Inmatning med scanf-funktionen

Inmatning från tangentbordet går till så att alla tecken man skriver samlas i en buffert, *stdin*, samtidigt som de också ekas på skärmen. När man matat in färdigt trycker man på RETURN och då scannar (läser och omvandlar) programmets scanf-sats bufferten och flyttar över data till angiven plats i minnet.

Ex: Inmatning av ett heltalsvärde till variabeln tal från tangentbordet.

```
#include <stdio.h>

int main()
{
   int tal;

   printf("Ge ett heltal : ");
   scanf("%d", &tal);

   return 0;
}
```

En körning:

Ge ett heltal : <u>234</u>

Vid körning stannar programmet då den kommer till scanf-satsen och väntar på att den som kör programmet ska skriva in ett heltal och trycka RETURN. När man gjort detta scannar scanf-satsen inmatningen enligt *formatsträngen* `"%d"`, som ovan endast inne--håller en *omvandlingsspecifikation* `%d`, vilket anger att tecknen i bufferten ska *tolkas som ett decimalt heltal* och talet ska sedan placeras i minnet *kodat som en int* på den adress som tal har.

Man kan läsa in värden till flera olika variabler med samma scanf-sats. Scanf-satsen innehåller då flera variabeladresser och varje variabeladress ska ha en motsvarande omvandlingsspecifikation i formatsträngen

Ex: Inmatning av två heltalsvärden till variablerna a_tal och b_tal i en enda sats.

```
#include <stdio.h>

int main()
{
   int a_tal, b_tal;

   printf("Ge två heltal : ");
   scanf("%d%d", &a_tal, &b_tal);

   return 0;
}
```

En körning:

Ge två hela tal : <u>345 268</u>

Här måste man *separera* de inmatade talen med någon *tillåten separator* annars tolkar datorn de inmatade tecknen som ett enda tal och stannar och väntar på nästa tal. Som separatorer kan man använda blanktecken (som ovan), RETURN (styrtecknet '\n') eller TAB (styrtecknet '\t'). Programmet gör så att den börjar med att hoppa över alla eventuella separatorer som den hittar i början. Sedan läser den fram till nästa separator och placerar det lästa talet i den första variabeln. Proceduren upprepas sedan för nästa variabel o.s.v.

Inmatningsbufferten läses utgående från formatsträngen. Har man exempelvis angivit %d innebär detta att programmet läser tecken så länge de passar in i ett decimalt heltal. Skulle den stöta på ett tecken som ej passar in, exempelvis bokstaven A, avbryts inläsningen och det dittills godkända tecknen omvandlas till heltal och placeras i variabeln. Hittas inga godkända tecken avbryts inläsningen direkt utan att något lästs in.

Ex: Några körningar av programmet ovan med felaktiga inmatningar.

a) Ge två heltal : <u>12S 321</u>

Detta ger a_tal = 12 och b_tal har kvar sitt gamla odefinierade värde.

b) Ge två heltal : <u>123 3F21</u>

Detta ger a_tal = 123 och b_tal = 3.

OBS! De tecken som ej accepteras blir kvar i inmatningsbufferten och kan ställa till besvär vid en eventuell ny inläsning. Man bör därför alltid se till att man har en tom inmatningsbuffert efter varje inläsning. I vissa lägen bör man även tömma bufferten på eventuella kvarvarande separatortecken. (Mer om det senare.)

Man kan med scanf-funktionen även läsa in reella tal och tecken. Dock kan man inte läsa in egenuppräknade variabler.

Ex: Skriv ett program som läser in ett additonsuttryck bestående av två reella tal enligt exempelvis 4.0+3.2 och beräknar och skriver ut resultatet.

```
#include <stdio.h>

int main()
{
   double a_tal, b_tal, sum;
   char op;

   /* läs in uttrycket */
   printf("Ge uttrycket på formen x+y : ");
   scanf("%lf%c%lf", &a_tal, &op, &b_tal);

   /* skriv resultatet */
   if ( op == '+' )
   {
      sum = a_tal + b_tal;
      printf("Resultat = %.1f\n", sum);
   }
   else
      printf("Fel operator!\n");

   return 0;
}
```

En körning ger följande :

Ge ett uttryck : 3.2+4.0
Resultat = 7.2

Formatsträngen "%lf%c%lf" i scanf-funktionen läser först fram till +-tecknet eftersom 3.2 kan omvandlas till ett reellt tal. Därefter läses ett tecken nämligen +-operatorn och slutligen det andra reella talet.

OBS! Omvandlingsspecifikationen %c hoppar ej över några separatorer utan läser varje tecken i inmatningsbufferten. Hade man exempelvis skrivit ett blanktecken före +-tecknet skulle op blivit ett blanktecken och programmet skriver "Fel operator!". Vill man hoppa över eventuella blanktecken får man ange detta i formatsträngen med "%lf %c %lf"och då fungerar en inmatning av typen :

Ge ett uttryck : 3.2 + 4.0
Resultat = 7.2

2.3.2 Utmatning med printf-funktionen

Utmatning till skärmen sker med printf-funktionen som fyller bufferten *stdout*, som sedan skickas ut till skärmen på den plats där markören står. Hur bufferten ska fyllas ska framgå av formatsträngen i printf-funktionen.

Ex: Inmatning av ett heltalsvärde till variabeln tal från tangentbordet och utmatning av det dubbla värdet.

```
#include <stdio.h>

int main()
{
   int tal, dubbelt_upp;

   printf("Ge ett heltal : ");
   scanf("%d", &tal);

   dubbelt_upp = 2 * tal;
   printf("Tal = %d\nDubbelt upp = %d\n", tal, dubbelt_upp);

   return 0;
}
```

En körning:

Ge ett heltal : 35
Tal = 35
Dubbelt upp = 70

Formatsträngen kan innehålla text, styrtecken och omvandlingsspecifikationer. Varje variabel ska ha en omvandlingsspecifikation som anger hur minnesutrymmet ska tolkas och skrivas ut. *Omvandlingsspecifikationen %d anger att minnesutrymmets kod ska tolkas som en int och skrivas ut decimalt i default format*. Texten i övrigt skrivs ut som den står förutom styrtecknen som skärmen reagerar för på ett speciellt sätt. Exempelvis flyttar styrtecknet '\n' skärmens markör till början på nästa rad.

Formatsträngens omvandlingsspecifikation kan även innehålla en formatering av utskriften där man anger antal positioner som utskriften ska skrivas ut på.

Ex: Vill man ha en utskrift vid körning ovan enligt:

```
Ge ett heltal :  35
Tal =            35
Dubbelt upp = 70
```

får man skriva formatsträngen så att utskriften sker högerjusterad i 12 positioner enligt:

```
printf("Tal = %12d\nDubbelt_upp = %d\n", tal, dubbelt_upp);
```

Utskriftsatsen kan innehålla mer sammansatta uttryck än enstaka variabler. Vill man bara skriva ut ett resultat av ett uttryck är det onödigt att skapa en extra variabel för detta, utan man kan både räkna ut uttryckets värde och skriva ut det direkt i `printf`-anropet.

Ex: Skriv ett program som läser in årsräntesats i % och kapital i kr och beräknar och skriver ut årsräntan i kr.

```
#include <stdio.h>

int main()
{
   double procent, kapital;

   printf("Årsränta i %: ");
   scanf("%lf", &procent);
   printf("Kapital i kr : ");
   scanf("%lf", &kapital);

   printf("Årsräntan blir %.2f kr\n", kapital*procent/100);

   return 0;
}
```

En körning:

Årsränta i % : 7.5
Kapital i kr : 1250
Årsräntan blir 93.75 kr

Istället för att använda en variabel för räntan beräknar man och skriver ut uttrycket med printf-funktionen. Formatsträngens omvandlingsspecifikation %.2f anger att minnesutrymmet, som här ej är namngivet med en variabel men innehåller det uträknade värdet, ska tolkas som ett reellt tal och skrivas ut med 2 decimaler.

Ex: Skriv ett program som läser in en liten bokstav a .. z och skriver ut motsvarande stora bokstav. I ASCII-tabellen, och i många andra teckenkodningar, är koden för en liten bokstav 32 enheter större än för motsvarande stora.

```
#include <stdio.h>

int main()
{
   char tecken;

   printf("Ge en liten bokstav : ");
   scanf("%c", &tecken);
   printf("Motsvarande stora bokstav är %c\n", tecken - 32);

   return 0;
}
```

Körning:

Ge en liten bokstav : b
Motsvarande stora bokstav är B

2.3.3 In- och utmatning av tecken med getchar- resp putchar-funktionen.

För att läsa in och skriva ut tecken kan man använda funktionerna scanf resp printf med omvandlingsspecifikationen %c. Det finns också speciella funktioner för just in- och utmatning av tecken. Dessa finns i *stdio.h* och heter *getchar* och *putchar*.

Funktionen getchar läser nästa tecken i inmatningsbufferten och returnerar tecknet som funktionsvärde, *alltså själva funktionsanropet motsvarar tecknet*. Funktionen putchar skriver ut det tecken som står mellan parenteserna, den så kallade parametern eller argumentet, på skärmen.

Ex: Skriv om ovanstående teckenomvandling från små till stora bokstäver och använd getchar och putchar istället för scanf och printf.

```
#include <stdio.h>

int main()
{
   char tecken;

   printf("Ge en liten bokstav : ");
   tecken = getchar();

   printf("Motsvarande stora bokstav är ")
   putchar(tecken - 32);
   putchar('\n');

   return 0;
}
```

Fördelen med att använda getchar och putchar är att man slipper formatsträngen med dess omvandlingsspecifikationer. Getchar tolkar alltid inmatningsbufferten som ett tecken och putchar skriver alltid ut den angivna parametern som ett tecken på skärmen.

OBS! De tomma parenteserna när en funktion anropas utan parametrar som i getchar().

Ex: Ibland måste man se till att inmatningsbufferten är tom för att nästa läsning ska fungera. Det som oftast finns kvar i bufferten är ett RETURN-tecken och då räcker det med att man anropar funktionen getchar en gång enligt:

```
getchar();
```

Har man även annat skräp före RETURN-tecknet måste man upprepat anropa getchar-funktionen enligt:

```
while ( getchar() != '\n' )
      ;
```

Här rensas alla tecknen från inmatningsbufferten fram till och med RETURN-tecknet. (Det enamma semikolonet på andra raden är en så kallad tom sats, som inte gör något.)

2.4 Uttryck

Ett uttryck består av operander och operatorer. Det enklaste uttrycket består av ett enda värde eller en variabel.

Ex: litet_tal + 23 är ett uttryck bestående av operanderna litet_tal och 23 och operatorn +.

Operander kan i sig vara uttryck.

Ex: x * (y - 3.2) är ett uttryck bestående av operanderna x och (y-3.2).

Alla utryck har ett värde av en viss typ av data. Vilken typ av värde ett uttryck får beror av de ingående operatorernas värden och typer. Man brukar skilja på *aritmetiska uttryck* som har värden i form av typen heltal eller reella tal och *logiska uttryck* som bara har två värden, sant eller falskt. I C är egentligen även logiska uttryck aritmetiska eftersom *falskt motsvaras av heltalet 0 och sant av alla andra tal*.

Ex:	pi*radie*radie	är aritmetiskt med ett värde i form av ett reellt tal
	tal < 0	är logiskt med ett värde 0 (falskt) eller 1 (sant)

2.4.1 Aritmetiska uttryck

Aritmetiska uttryck innehåller de vanliga operatorerna +, -, * och /. Dessutom finns en del speciella operatorer som % (resten vid division), ++ (öka med 1) etc.

Ex:	5 + 3	har värdet 8 av typen heltal
	5 / 2	har värdet 2 av typen heltal (kvoten vid heltalsdivision)
	(double)5 / 2	har värdet 2.5 av typen reellt tal (explicit typomvandling)
	5 % 2	har värdet 1 av typen heltal (resten vid heltalsdivision)
	5 + 3 * 2	har värdet 11 av typen heltal
	(5 + 3) * 2	har värdet 16 av typen heltal

OBS! När man blandar olika typer av operander i uttryck omvandlas först alla operander till den högsta typen varefter beräkning sker. Med högsta typ menas den som har störst minnesutrymme. Blandar man som i uttrycket (double)5 / 2 ett heltal med ett reellt tal omvandlas heltalet automatiskt till ett reellt tal varefter division utförs för reella tal.

OBS! Samma operator kan ha olika innebörd beroende på vilka datatyper som den ska verka på. Divisionsoperatorn / utför heltalsdivision (hur många hela gånger går den högra operanden i den vänstra) om den verkar på två heltal och vanlig reell division om den verkar på reella tal.

OBS! De vanliga matematiska prioriteringsreglerna gäller där /, * och % har högre prioritet eller utförs före + och -. Med parenteser ändras prioriteten så att parentesen utförs först. *Använd parenteser för att få tydligare kod!*

Efter beräkning av ett värde på ett uttryck sker ofta en tilldelning av detta värde till en variabel. Som tilldelningsoperator används likhetstecken.

Ex:
```
int a_tal, b_tal;

a_tal = 6 * 2 % 5;    /* a_tal får värdet 2 */
b_tal = 2.3 * 2;/* b_tal får värdet 4 */
a_tal =  b_tal  = 0;  /* b_tal och a_tal får båda värdet 0 */
```

OBS! Först beräknas det högra uttyckets värde och typ. Är det högra uttryckets typ samma som typen hos den vänstra variabeln sker tilldelning direkt. Är däremot typerna olika sker en *automatisk (implicit) typomvandling* av det högra uttryckets värde till den vänstra variabelns typ.

OBS! En tilldelning kan ses som ett uttryck med ett värde som är lika med det tilldelade värdet.

Tilldelningsoperatorn finns i att antal sammansatta former som +=, *= etc.

Ex:
```
double x = 2.3;
int a = 3;

x = x + 2  /* x blir 4.3 */
x += 2;    /* samma operation, x blir 6.3 */

a = a * 2; /* a blir 6 */
a *= 2;    /* samma operation, a blir 12 */
```

OBS! Istället för en tilldelningssats kan man initiera variabelvärden vid definitionen. Ger man inga värden vid definitionen är variabelvärdet odefinierat.

Ofta när man skriver programkod så ingår satser där man ökar eller minskar en variabels värde med ett. I C har man infört speciella operatorer för detta.

Ex: Skriv ett program som skriver ut alla stora bokstäver mellan A och Z.

```
#include <stdio.h>

int main()
{
   char ch = 'A';

   do
   {
      putchar(ch);
      ch++;           /* nästa bokstav */
   }
   while ( ch <= 'Z' );

   return 0;
}
```

Körning ger:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

När man arbetar med aritmetiska uttryck har man ofta användning av de vanliga matematiska funktionerna sin, cos, log, exp etc. Dessa finns i *math.h*.

Ex: Skriv ett program som frågar efter ett reellt tal och skriver ut kvadratroten ur talet och kvadraten på talet.

```
#include <stdio.h>
#include <math.h>     /* sqrt, pow */

int main()
{
   double tal;

   printf("Ge ett tal : ");
   scanf("%lf", &tal);

   printf("Kvadratroten ur talet = %f\n", sqrt(tal));
   printf("Kvadraten på talet = %f\n", pow(tal, 2));

   return 0;
}
```

Exempel på körning:

Ge tal : <u>5.3</u>
Kvadratroten ur talet = 2.302173
Kvadraten på talet = 28.090000

Ex: och Skriv ett program som frågar efter en rätvinklig triangels hypotenusa och en vinkel beräknar triangelns övriga sidor.

```
#include <stdio.h>
#include <math.h>     /* sin, cos, atan */

int main()
{
   double hypotenusa, vinkel;
   const double pi = 4*atan(1);

   printf("Ge hypotenusan och vinkeln : ");
   scanf("%lf%lf", &hypotenusa, &vinkel);

   printf("Ena sidan = %f\n",hypotenusa*sin(pi*vinkel/180));
   printf("Andra sidan = %f\n",hypotenusa*cos(pi*vinkel/180));

   return 0;
}
```

Körning:

Ge hypotenusan och vinkeln : <u>35.6 30</u>
Ena sidan = 17.800000
Andra sidan = 30.830504

OBS! Vinkeln måste skickas i radianer vid anrop av de trigonometriska funktionerna.

2.4.2 Logiska uttryck

Ett logiskt uttryck innehåller ofta någon av *relationsoperatorerna* < (mindre än), > (större än), == (lika med), != (skilt ifrån), <=(mindre än eller lika med) och >= (större än eller lika med). Värdet av ett logiskt uttryck är falskt (dvs talet 0) eller sant (alla andra tal). Värdet av en jämförelse med en relationsoperator blir alltid 0 (falskt) eller 1 (sant).

Ex:		
	3 < 5	har värdet 1 och typen heltal
	5 == 2	har värdet 0 och typen heltal
	5 != 2	har värdet 1 och typen heltal
	3 >= 6	har värdet 0 och typen heltal
	3 + 4 <= 2 + 6	har värdet 1 och typen heltal

OBS! Relationsoperatorerna har lägre prioritet än de vanliga aritmetiska operatorerna. Man behöver ej prioritera med parenteser i 3 + 4 <= 2 + 6. För tydlighetens skull kan man dock göra detta med (3 + 4) <= (2 + 6).

Flera logiska uttryck kan sammanfogas med de *logiska operatorerna* && (och), || (eller), ! (inte).

Ex:		
	3 < 5 && 5 == 2	har värdet 0 eftersom det krävs att båda uttrycken är sanna
	3 < 5 || 5 == 2	har värdet 1 eftersom det räcker om ett uttryck sant
	! (3 < 5)	har värdet 0 eftersom inte sant är falskt

OBS! De logiska operatorerna && och || har lägre prioritet än relationsoperatorerna medan ! har högre. Är man osäker på prioriteten ska man använda parenteser. Det gör inget om man har några onödiga parenteser. Det ökar bara tydligheten.

Logiska uttryck förekommer ofta i villkoren för upprepningar (while) och val (if) i programkoden.

Ex: Skriv den del av ett program som inte ger sig förrän ett korrekt månadsnummer mellan 1 och 12 har inlästs.

```
....
printf("Ge ett månadsnummer mellan 1-12: ");
scanf("%d", &mm);
while ( mm < 1 || mm > 12)
{
   printf("Fel månadsnummer! Vi gör ett nytt försök!\n");
   printf("Ge ett månadsnummer mellan 1-12: ");
   scanf("%d", &mm);
}
....
```

I *ctype.h* finns speciella funktioner för att hantera tecken. Många av dessa funktioner är av typen att man ska kontrollera om ett tecken exempelvis är skrivbart, siffra etc. Dessa funktioner returnerar ett logiskt värde sant eller falskt och kan alltså sägas vara logiska uttryck.

Ex: Skriv ett program som läser in ett antal tecken som avslutas med RETURN-tecknet och som skriver ut alla tecken som är siffror.

```
#include <stdio.h>
#include <ctype.h>    /* isdigit */

int main()
{
   int ch; /* Ja, "int" och inte char"! */

   printf("Skriv en rad med tecken: ");
   ch = getchar();
   while ( ch != '\n' )
   {
      if ( isdigit(ch) )
         putchar(ch);
      ch = getchar();
   }

   return 0;
}
```

Körning:

Skriv en rad med tecken : Abjhacjj1332dasdd2114dd212
13322114212

Man ser ofta i C-program att man skriver ovanstående kod på ett förkortat sätt genom att utnyttja att *tilldelningar får värdet av det tilldelade uttrycket*. Ovanstående kod kan istället skrivas som:

```
#include <stdio.h>
#include <ctype.h>    /* isdigit */

int main()
{
   int ch; /* Ja, "int" och inte char"! */

   printf("Skriv en rad med tecken: ");
   while ( (ch = getchar()) != '\n' )
   {
      if ( isdigit(ch) )
         putchar(ch);
   }

   return 0;
}
```

OBS! Vill man skriva flera rader av tecken får man istället avsluta inskrivningen med en filslutsmarkering (som kan skrivas med CTRL-Z i Windows och CTRL-D i Unix), och i programmet testa på EOF istället för RETURN-tecknet '\n'.

OBS! Notera att det tecken som `getchar` läser in bör lagras i en `int`-variabel, och inte i en `char`-variabel! Fråga läraren om varför!